An Architecture for Reference Linking

Donna Bergmark, William Arms* and Carl Lagoze†
Cornell Digital Library Research Group

October 2, 2000

Abstract

The Digital Library Research Group at Cornell has Reference Linking as one of its projects. Typical projects within in the group take an object-oriented approach to handling digital information. To support reference linking, therefore, we designed a scheme whereby reference linking information is extracted from archives by *surrogate* objects and then presented to client applications or users by means of a well-defined API. This paper describes that architecture, the API, and how the API might be supported in the Dienst protocol.

1 Background

This report covers the design of a new reference linking service, supported by a DARPA/CNRI grant to intralink D-Lib and a JISC/NSF grant to intralink the Los Alamos physics E-prints archive, now called arXhive. The ultimate goal of this work is not only to intralink these two archives of online literature but to interlink Open Archives[14] and some online journals.

What does it mean to "intralink" and "interlink"? Reference Linking is actually an old idea. Classical reference linking arose from a desire to study citation patterns among scholarly articles. The Science Citation Index, founded by Eugene Garfield in the 70's, was invented to do just that, and was a spectacular success. It was, however, based

*DARPA/CNRI Grant #2057/57-02
†NSF Grant # IIS-9907892

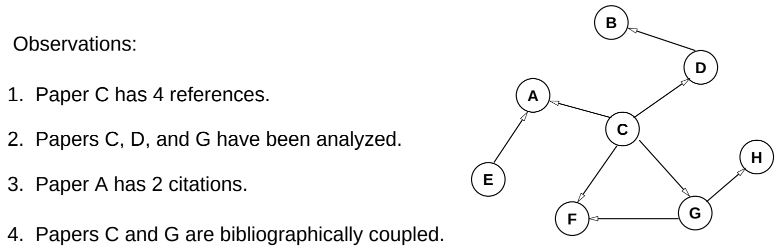

Figure 1: Classical Reference Linking

on human labor. For every paper examined, the staff captured that paper's metadata, and then went to the reference section and did the same for each reference there, or at least for those references to journals covered by the SCI.

As a result, one could look up links using the Science Citation Index and build a graph as shown in Figure 1. From this graph we can observe that Paper C has 4 references, that Papers C, D, and G have been analyzed, that Paper A has two citations, and that Papers C and G are bibliographically coupled (i.e. they have a reference in common). The links in the graph are *explicitly* contained in the Science Citation Index.

We then fast-forward some 25 years to the current time, where there is a growing amount of scholarly literature online. Much of this has HTML links to other works on the web. As in classical reference linking, the references are inserted by authors. Some are accompanied by URLs, but not all.

Unlike SCI and classical reference linking, citations cannot be directly discovered from the Web. It is a daunting task to analyze items on the web to find out who might have cited a paper of interest.

Figure 2 shows how interlinked papers on the Web might exist. The graph is implicit, defined by links between papers. It is likely to be quite large. From the fragment shown here, we can deduce the HTML page C has four links in it to other HTML pages; page A has at present two links to it; and papers C and G are linked to a common page. But, discovering this fragment from traveling the web is nearly impossible. The graph exists *implicitly* on the Web.

In our reference linking project we are aiming somewhere between the classical view and what exists today on the web. We wish to make the graph in Figure 2 explicit, as well as supply additional links where possible. By making the links explicit, new applications are possible.

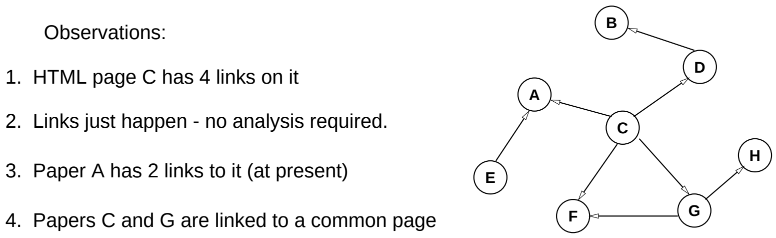

Figure 2: Linking on the Web

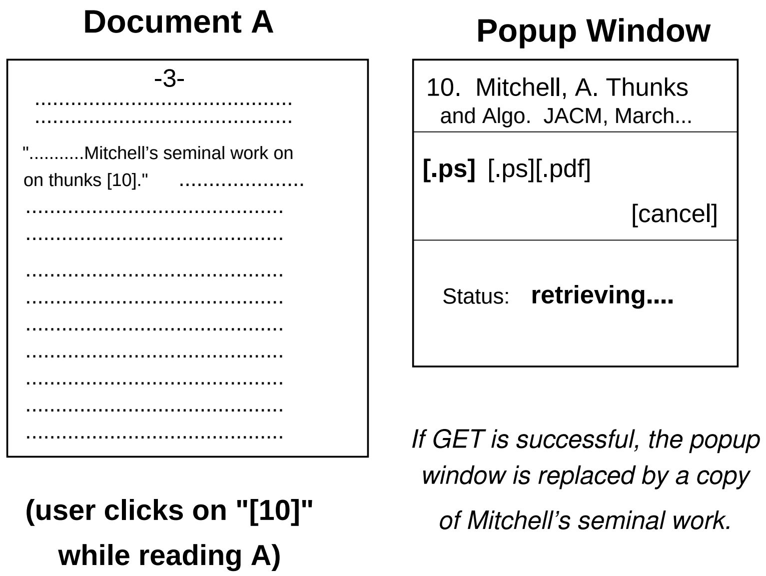

Figure 3: A Reference Linking Application

Figure 3 is just one example of a reference linking application.

You are reading a paper on the screen (or hearing it on your speakers, etc.) and you come across an intriguing reference: "...Mitchel's seminal work on thunks[10]." If there is a copy of this work somewhere online, the "[10]" would be turned into a clickable live link, so that the user could start fetching that copy while continuing to read the original paper. One interface that would support this goal might be a JavaScript popup window that looks something like the one on the right side of Figure 3; the complete reference string is shown along with some choices of format (PostScript, PDF) in which the document might be retrieved; the user can retrieve one of these or cancel.

Note that we might have not only static links as shown in the popup window, but we might add some links dynamically, say by running a Google search in the background, which might add a fourth option to the current `.ps` and `.pdf` options.

Implementing the functionality shown in Figure 3 requires solving at least two problems: 1) Figuring out that "[10]" is a reference and that it matches the reference string, [*10*] *Mitchell, A. Thunks and Algol...*; then parsing *Mitchell, A.* to decide what work it is and whether it is linkable (this is a *tough* problem!); and deciding whether it is something we've seen before so we can credit Mitchell with a citation.

2) Turning the "[10]" into a live link. In HTML and PDF you can turn this into an anchor that can be clicked. For other formats some kind of auxiliary display is needed. In fact, it is probably best just to record bibliographic data with the anchor, and let a translation system, such as XSLT, turn it into a link of the desired form.

In any case the first problem is one of *analysis* and the second is a *presentation* problem. The API, to be described in this paper, is responsible for supplying sufficient data for value added services, such as creating live links.

2 Definitions

The previous section was a quick introduction to reference linking. In this section we present some basic terms and definitions, so that we can explore the problem in more detail.

2.1 Items and Works

In Figure 3, call the lefthand side thing A and call the line [*10*] *MitchelA. Thunks...* on the righthand side thing B. There is a subtle, but important, difference between A and B. A is an *Item*, something that has a format, something that is online, something that can be analyzed with a computer program. B, which is a one of the references in A, is a *Work*, or an abstraction of a paper. Zero or more copies of it may in fact exist. Some of them may even be online. We say that A *references* B. B is one of A's references. It probably even shows up in a section of A titled `References`.

The words we just defined - *Work* for the abstract paper and *Item* for a concrete instance of that paper – is taken from the IFLA model [8, 12], shown in an abbreviated form in Figure 4. Since in dealing

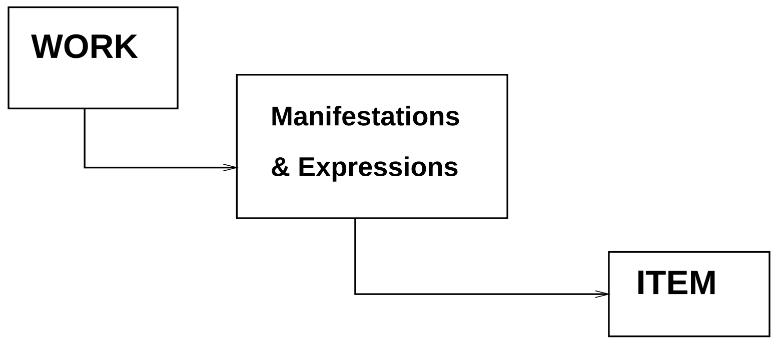

Figure 4: IFLA Model

with online stuff one only cares about Works and Items, we ignore the middle two levels. In our experience with scholarly and scientific work, the reference is usually to the Work (rather than to, say, a specific instance of that work held in a particular collection, in a particular format). See Svenonius [13] for a good philosophical discussion of what a work is.

In the rest of this report, we drop the capitalization of work and item, but continue to distinguish between the two terms.

2.2 References and Citations

Going back to Figure 3, item A references work B, an abstraction of a paper by Mitchell. If in fact, a copy of Mitchell's work can be found online, then it is a *linkable reference*. A *citation* is sort of the inverse of reference. Here, the abstract paper of which A is an instance is a citation of B. Tracking citations is not immediately needed for reference linking, but is a tempting addition to any reference linking service. There is, of course, a subtle difference in tractability of making, for an item, its list of references and its list of citations. The labor involved in finding the citations is what made the SCI a huge success. The main thing to note here is that like references, citations are works.

With this background in reference linking, we are ready to say more about the approach being taken by the Cornell reference linking project.

3 The Reference Linking API

The reference linking architecture developed at Cornell is unusual in several respects. Most other reference linking projects (e.g. Open Journals[4] and ResearchIndex[6]) use databases to store information about works and do a lot of "database crunching". For example, there would be one database of all the titles, and perhaps another database of all the authors, and a third with references. Instead of using databases to store this information, we use item surrogates. A *surrogate* is a digital object that encapsulates reference linking information relating to an item that is being analyzed. Reference linking data is thus distributed across the collection of surrogate objects, and all the data relating to one item is grouped together within the surrogate for that item.

Reference linking is amongst the set of extensible behaviors we would like to add to digital objects, as pointed out by Payette [10]. In general, we refer to such objects as "value-added surrogates".

Another unusual aspect of reference linking at Cornell is that we define an API for reference linking. Having an API specifies the operational semantics of reference linking; it also allows us to cleanly separate the analysis phase of reference linking from the presentation phase. For example, one method in the API is *getReferenceList*, which returns harvested metadata for each reference contained in an archive item, such as its title, publication, context in which it was cited, year and authors. This data, encoded as XML data, is suitable for further processing by other applications. The advantage of creating an API is that no decision is made in advance of what the data should be used for; our work is not wrapped up into a single stand-alone reference-linking service as is the case in other projects.

The combination of surrogates and an API essentially allows us to walk up to a paper and ask it "what are your references" and "is reference 10 linkable?" One surrogate is constructed for each archive item, and can answer questions about that item. The set of questions it is prepared to answer *is* the API. Each surrogate answers the same set of questions (to the best of its ability).

This architecture is depicted in Figure 5. The central column represents some repository of network-accessible documents. The items listed in this column are linkable (they are online) and therefore analyzable (we have their Bits).

On the left are drawn the works that the items represent. Any

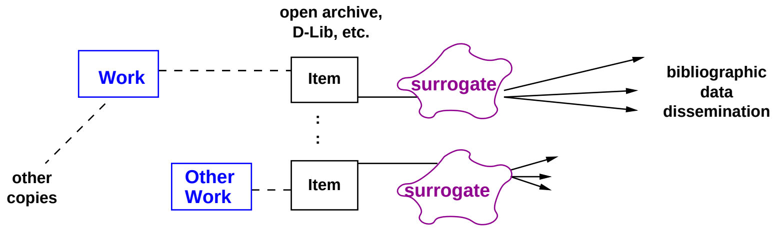

Figure 5: An Architecture for Reference Analysis

work might have several copies spread across several archives. All of these copies are "items" corresponding to that work. If more than one copy of a work is encountered, we could pool the information collected so that both surrogates have consistent data, but this requires either that the surrogates be able to find and communicate with each other, or that there be a central database. Arranging for the surrogates to communicate among themselves is an interesting research problem; for now we keep a small database of works seen so far which at least allows sketchy information to be updated. To the right of the archive items are the surrogates, shown as blobs. They know how to disseminate the answers to the questions asked of them, via the API. The answer is the appropriate dissemination of bibliographic data about the item, and indirectly, about the work. As stated above, client applications ask the questions and then display or otherwise use the results. The API supplies the data.

Surrogates could answer a number of different questions, but the four primary questions in the Cornell reference linking API are:

- `getLinkedText` – contents of the paper (as data) augmented with reference linking data. This question would be asked by browsers that wanted to display the document with some of its references turned into anchors of live links, as in Figure 3.
- `getReferenceList` – this interface would be used by applications that wish to know what references are contained in this paper. For example, if one were building the SCI, this would be the question to ask, along with the next one.
- `getMyData` - this returns that paper's metadata. This is not directly related to reference linking, but is required for building up citation relationships. It could have other uses; for example, one client might have a button labeled "get BibTeX"; when the

button is pushed, the client invokes `getMyData` on the surrogate, and reformats the results into something suitable for cutting and pasting into a LaTeX bibliography.

- `getCurrentCitationList` – the list of works citing this paper to the best of the surrogate's knowledge. As stated before, this function is not strictly required for reference linking, but would be very useful to client applications that want to know what other documents cite this one, as they might be related or provide more current information. If online, we have a *linkable citation*.

3.1 Output from the API

Figure 5 shows the surrogates disseminating bibliographic information about their items, in response to a particular method in the API being invoked. Each method returns a byte-stream of structured data in the form of an XML file, to permit further processing. Figure 6 shows what one component of the XML information disseminated by `getReferenceList` might look like. This component is the second reference (`ord="2"`) for this surrogate's item.

First comes bibliographic data related to the reference work. We are using Dublin Core for convenience, so for example, dates must be in `CCYY-MM-DD` format.

Next there is item-related information, such as the reference string exactly as it appeared in the item (enclosed in a `<literal>` element and entified), and all the contexts in which the work was cited. The context is usually one complete sentence, as shown near the bottom of Figure 6. Note the "[2]" in the context. Since the Maly paper does have a URL, this may become the anchor of a live link in any text returned by a call to `getLinkedText`.

3.2 Various Specifications of the API

Up to now, the API has been described in rather general terms with a detailed description of its output. There are many different ways to implement the API. For example, figure 7 shows ways to implement `getReferenceList`.

If we were to implement the method in Java (and we are), the method would be described in terms of a Java signature and perhaps the procedural code attached to this signature (the implementation is

```
<api:reference_list length="17"
      xmlns:api="http://www.cs.cornell.edu/cdlrg/..."
      xmlns:dc="http://purl.org/DC">
    :
    :
<api:reference ord="2">
<dc:title>
Smart Objects, Dump Archives: A User-Centric, Layered Digital
Library Framework
</dc:title>
<dc:date>1999-03-01</dc:date>
<dc:identifier>10.1045/march99-maly</dc:identifier>
<dc:creator>K Maly</dc:creator>
<api:displayID>
http://www.dlib.org/dlib/march99-maly/03maly.html
</api:displayID>
<api:literal tag="2.">
Maly K, "Smart Objects, Dumb Archives: A User-Centric, Layered Digital
Library Framework" in D-Lib Magazine, March 1999,
&lt;http://www.dlib.org/dlib/march99-maly/03maly.html&gt;.
</api:literal>
<api:context list>
<api:context>
The need for standards to support the interoperation of digital library
systems has been reported on before in D-Lib[1],[2] as have efforts to
discover common ground in related standard processes(Dublin Core and
INDECS[3]).
</api:context>
</api:context list>
</api:reference>
   :
   :
</api:reference_list>
```

Figure 6: XML for a Reference Object

API	getReferenceList
Java	`Byte[] getReferenceList()`

Dienst	`Verb: Disseminate`
	`Version: 1.0`
	`FixedArgs:FullID,getReferenceList,xml`
	`KeywordArgs: (none)`
	`Return MIME type: text/xml`
	`Return Status Codes: 200, 400, 404`
	
	`Example request:`
	`Dienst/Repository/1.0/Disseminate/10.1045/`
	`december99-miller/getReferenceList/xml`

Figure 7: Various Specifications of the API

a very detailed form of specification). This would be one of the public methods of the Surrogate class.

If we were to implement the API in Dienst[5] (and we are), then we would lay out what for `getReferenceList` the Dienst protocol should look like. The `disseminate` verb takes three fixed arguments, the second of which is the *view*. Views in Dienst are like particular disseminations of a document; here the protocol is requesting an XML-formatted list of references.

An example HTTP request is shown at the bottom of the table. The result of this request would be a Dienst response (verb version number, etc.) plus the XML listed in Figure 6.

This report will next look at the Java implementation and the Dienst implementation in greater detail.

4 Java Implementation: the Classes

In Java, classes are templates for what the objects should know and do. For the Java implementation of the reference linking API, the main place to start is to define a Surrogate class. The public methods of this class define the API. Given the four main methods in the API (see Section 3), Figure 8 shows the series of Java objects that emerges.

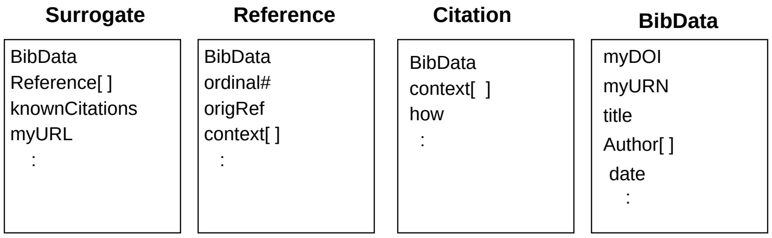

Figure 8: API in Java: The Objects

The Surrogate is, of course, the principle object. It contains just a few important fields. The first is a BibData object, which contains information about the work corresponding to the item for which this object is the surrogate. The BibData object contains typical work-related metadata. The key, "myURN" is the API's identification of this work and is constructed out of metadata belonging to the work. It is a form of URN, and meets most of the requirements laid out by Paskin [7].

The Surrogate contains a fixed-size array of Reference objects. The Reference object consists of a BibData object (metadata for the reference), the literal reference as it appeared in this item, and the list of contexts in which it was referenced in this item.

The Surrogate also contains a vector of Citation objects. This is a vector rather than an array, so that it can grow in length as other surrogates are created for items that cite the work related to this surrogate. The Citation object contains bibliographic data relating to the cited work and to the circumstances of the citation.

Finally the Surrogate knows the URL, or location, of its item. A client reference linking application always accesses the item through its surrogate.

4.1 Implementation Details

This section is a detailed discussion of our Java implementation and may be skipped by the reader who is more interested in just an overview of the reference linking API.

The Java implementation of the reference linking API exists of three packages, as depicted in Figure 9.

The first package contains the public interface to the reference linking software, and is for client applications. Client applications

package	purpose	lines
Linkable.API	This is what the client application uses.	875
Linkable.Analysis	Contains a set of parsers and analyzers.	2500
Linkable.Utility	Static database functions and utilities.	1350

Figure 9: API in Java: Three Packages

instantiate a surrogate and hand it a URL of an item to be analyzed. Depending on the format of the item at the other end of the URL, the surrogate instantiates a parser. For example, one of the parsers that could be instantiated is `HTMLAnalyzer`, for parsing HTML documents. Each parser is an implementation of the `RefLinkAnalyzer` interface.

This interface is a key element of the `Linkable.Analysis` package. Reference link parsers currently have three required methods: `buildLocalMetadata()`, `buildRefList()`, and `buildCitationList()`. These methods are invoked by the surrogate in order to populate its private data fields.

When parsing HTML files, we found it useful to convert HTML items to XML (i.e. XHTML) before attempting to parse them[1]. Therefore when the surrogate is handed an HTML document to analyze, it first invokes JTidy [11] to convert the document to XML form and then instantiates an `XHTMLAnalyzer` to parse it. Another implementation of `RefLinkAnalyzer`, `RiggedAnalyzer`, knows all the information for a particular paper. This could be used for testing presentation clients.

The `Linkable.Utility` package has all responsibility for database construction and maintenance, as well as a number of static utility routines. (The Java implementation of the API keeps around a few hashtables for keeping track of works, authors, and citations.) This package also contains methods for DOM/SAX parsing of XML files and generation of XML output. It has some routines for parsing author names.

4.2 Sample Application

The API Java implementation is being currently being used for analyzing D-Lib articles. D-Lib is an on-line journal that has been appearing eleven times a year since July 1995; it makes an excellent test bed because there is little editorial imposition on the format of the papers

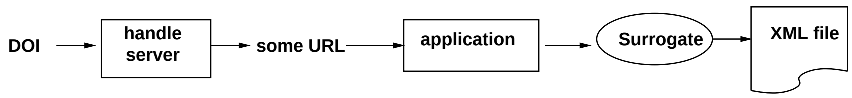

Figure 10: A Simple Reference Linking Application

submitted to the journal, and therefore provides a wide selection of paper layouts.

In Figure 10, the client application is given the URL of a D-Lib paper. It opens a connection to that URL, and then constructs a surrogate object, passing it that URL. All further interactions with the reference linking API are via this surrogate. In the real world, the application might instead be handed a DOI, and then use a handle server[2] to get a URL.

The right-hand side of Figure 10 tries to show that the client application, having instantiated a surrogate for the item to be analyzed, can now invoke various methods on the surrogate. The result of any of these invocations is an XML byte array, which can then be written to a file, or presented to the user in some way.

Figure 11 is a code snippet that represents the portion between the dotted lines in Figure 10. Following are some key things to note:
(1) The program starts by importing the `Linkable.API` package, which implements the methods contained in the reference linking API. The package includes the Surrogate class.
(2) The main program executes in three steps: it calls `initialize` to open a file called "D-LibArticles", which contains the URLs of some D-Lib papers; it then calls `createSurrogates` which constructs a surrogate for each item; and then it calls `finalize` to clean up.
(3) The `CreateSurrogates` routine executes a loop that reads a URL from the file, constructs a surrogate for it:

```
s = new Surrogate( url );
```

and then prints out its list of references:

```
System.out.println(s.getReferenceList().toString());
```

In real life, the surrogate object would not be created and thrown away; it would be saved in some form of persistent storage so that could be used again.[1] FEDORA [9] is one such persistent store. A

[1]This means that in fact the API contains methods for storing and reloading surrogate objects.

```
import Linkable.API.*;
import java.io.*;

public class DLIB {

   private static final String FILENAME = "./D-LibArticles";
   private BufferedReader in = null;

   public static void main (String[] args) {
      initialize ();
      createSurrogates();
      terminate();
   }
   :
   private static void createSurrogates() {
      String url;
      Surrogate s = null;

      try {
      while ( (url = in.readLine()) != null ) {
         s = new  Surrogate( url );
         System.out.println(s.getReferenceList().toString());
      }} catch (IOException e){ System.exit(0); }
   }
```

Figure 11: A Simple Application Program

```
Dienst Request:      list-contents

Possible Response:

      <?xml version="1.0">
      <List-Contents version="4.0">
      <record>arXiv:hep-th/9801001</record>
      <record>arXiv:hep-th/9801002</record>
      </list-contents>
```

Figure 12: Request-Response Protocol Specification

FEDORA object could encapsulate the data part of a surrogate object and store it in a distributed FEDORA repository.

5 Embedding the API in Dienst

Dienst is an architecture, a protocol, and a set of software that can be used to store and retrieve online documents. We can impelment the reference linking API in Dienst by specifying the protocol for talking with a surrogate. While APIs specify signatures – i.e. names of procedures, types of arguments, and the value returned – protocols accomplish that same thing by specifying the syntax for sending requests and responses over the network.

Figure 12 is an example of a simple Dienst request and response, according to the current Dienst protocol. In this example, "list-contents" is a way of asking an archive what items it contains. It returns a set of URNs. Note that all Dienst responses are in XML.

As mentioned before, we can extend the existing Dienst protocol with methods from the reference linking API by encoding them as views. For example, requesting the getReferenceList view will – if there is a surrogate for this item – return the same `byte[]` stream as returned by the corresponding call in Java, except that at the beginning there is some Dienst-specific XML output, such as the verb version that is being used.

The four main API methods of Section 3 (getLinkedText, getReferenceList, getMyData, getCurrentCitations) become views of the Disseminate verb. The Dienst protocol for learning what views can be disseminated for any given document is the Structure verb. Thus we need only add four new views to the structure verb; for example, one possible response to the structure request, assuming that our item has a surrogate, is:

```
<view id="getReferenceList" nrefs="15">
```

How does a Dienst implementation know whether a particular item is reference-linked? In NCSTRL [3], one popular implementation of the Dienst architecture, each item is a directory containing several different formats of the paper, along with other data. One could simply add another subdirectory called "Surrogate" inside the same directory. The presence or absence the the directory determines whether or not the four additional views of the API are available.

The contents of the Surrogate subdirectory are the data required to reconstitute the surrogate into an object able to disseminate reference linking information.

6 Architecture Recap

Up to this point, we have discussed the API: its methods and outputs. However, recall that the application shown in Figure 3 requires solving two very difficult problems: analysis, to find the live references; and presentation of those live references. Figure 13 shows this two-phased architecture of analysis and presentation. The interface between the two phases is structured data – XML. This report has not dealt with the presentation problem, but the overall architecture has been discussed.

On the left of Figure 13 is an archive to be analyzed (or which has been analyzed already). The databases save metadata sufficient to reconstruct surrogates. Whether the surrogate has been instantiated or reconstituted, it can then be used to generate an XML file. Figure 13 shows the response to "`getLinkedText`". This XML along with an XSL stylesheet can be converted into a document suitable for use by a browser or a renderer, or into a display similar to the one shown in the popup box in Figure 3.

Linkable references in the text become anchors to links sufficiently rich to point to various on-line copies of the reference, to retrieve the reference string itself, and so on. For example, the link could be an XLink, as shown in Figure 14. Here, `<ref-xl ... other stuff ...>` is a reference link element. The `xmlns` being used here is the standard

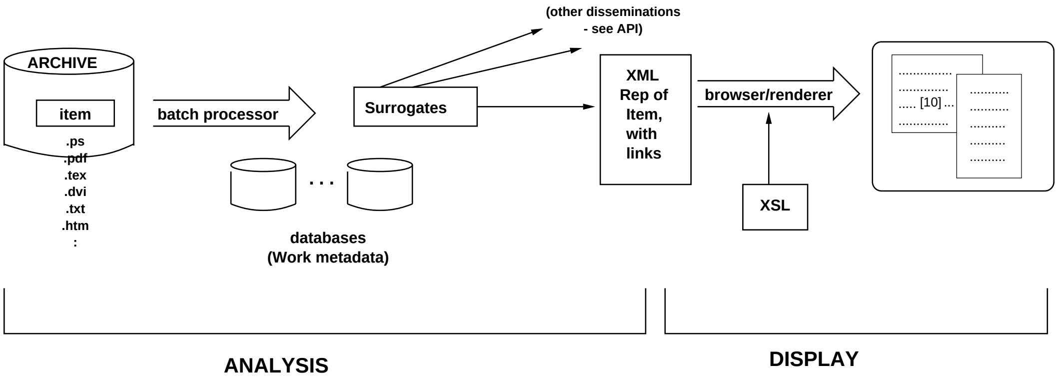

Figure 13: Overall Reference Linking Architecture

.

```
Original Text:
               ... in D-Lib[1],[2] ...
Linked Text:
   ...in D-Lib<ref-xl xmlns:xlink="http:..."
  xlink:type="simple" xlink:href="http:..dlib..">[1]</ref-xl>,
  <ref-xl xmlns:xlink="http:..link"
  xlink type="simple"
  xlink:href="uri:10.1045/march99-maly">[2]
  </ref-xl>...
```

Figure 14: An XML Item, with simple XLinks. Two references, 1 and 2, become anchors of standard URLs.

one, `http://www.w3.org/1999/xlink`. The simple type of XLink is just a URL.

7 Project Status and Conclusions

This project is well underway. The design of the API is done, and the Java implementation is in progress. The Dienst implementation has been designed.

The main difficulty, as everyone knows, is parsing text that has been produced by many different authors in many different formats with many different conventions. A separate paper [1] discusses this problem in more detail, and presents some algorithms useful in reference linking.

We are currently analyzing the papers in D-Lib for reference linking information. The D-Lib papers are all written in HTML. Only since 1999 are the papers accompanied by metadata. As a consequence, we are picking up almost all of the information directly from the text.

At this point we are analyzing papers, examining the errors, patching up the API Java code, and then analyzing new papers. With each additional paper that gets processed, the API implementation improves a little. Starting with one perfectly-analyzed paper, the next randomly chosen 10 came up with about 50% perfect extraction of metadata about the item itself, but only 40% perfect parsing of the available references. At the present time, those numbers both stand at 50%. If we look at the *proportion* of elements that can be extracted from an item or from a reference, we have 79% item accuracy and almost 80% reference accuracy. The work done so far indicates that the architecture and design for the reference linking API are sound.

References

[1] Donna Bergmark. The reference linking problem. Technical report, Cornell Computer Science Department, September 2000. in preparation.

[2] Priscilla Caplan and William Arms. Reference linking for journal articles. *D-Lib Magazine: The Magazine of Digital Library Research <http://www.dlib.org>*, 5(7/8), July/August 1999. <http://www.dlib.org/dlib/july99/caplan/07caplan.html>

[3] James Davis and Carl Lagoze. NCSTRL: design and deployment of a globally distributed digital libra ry. *IEEE Computer*, February 1999.

[4] Steve Hitchcock, Les Carr, Wendy Hall, Stephen Harris, S. Probets, D. Evans, and D. Brailsford. Linking electronic journals: Lessons from the Open Journal project. *D-Lib Magazine: The Magazine of Digital Library Research <http://www.dlib.org>*, December 1998.

[5] C. Lagoze and J. Davis. Dienst: An architecture for distributed document libraries. *Communications of the ACM*, 38(4):47, April 1995.

[6] Steve Lawrence, C. Lee Giles, and Kurt Bollacker. Digital libraries and autonomous citation indexing. *IEEE Computer*, 32(6):67–71, 1999. <http://www.researchindex.com>

[7] Norman Paskin. DOIs and reference linking, February 1999. A presentation to the NISO/NFAIS/SSP Linking Workshop, available online at http://www.niso.org/paskin.html

[8] Norman Paskin. E-citations: actionable identifiers and scholarly referencing, 1999. <http://www.doi.org/citations.pdf>

[9] S. Payette and C. Lagoze. Flexible and extensible digital object and repository architecture (FEDORA). In *Second European Conference on Research and Advanced Technology for Digital Libraries*, Heraklion, Crete, 1998.

[10] Sandra Payette and Carl Lagoze. Value-added surrogates for distributed content. *D-Lib Magazine: The Magazine of Digital Library Research <http://www.dlib.org>*, 6(6), June 2000.

[11] Andy Quick. Java HTML tidy. <http://www3.sympatico.ca/ac.quick/jtidy.html>

[12] K. G. Saur. Functional requirements for bibliographic records, 1998. UBCIM Publications - New Series Vol. 19.

[13] Elaine Svenonius. *The Intellectual Foundation of Information Organization*. M.I.T. Press, 2000.

[14] Herbert Van de Sompel and Carl Lagoze. The Santa Fe Convention of the Open Archives Initiative. *D-Lib Magazine: The Magazine of Digital Library Research <http://www.dlib.org>*, 6(2), February 2000.